《openEuler内核编程》

课程讲稿

第三章 第2讲

页式内存管理机制

软件所制

第三章 第2讲 页式内存管理机制

**学时：**2学时

**教学目的：**

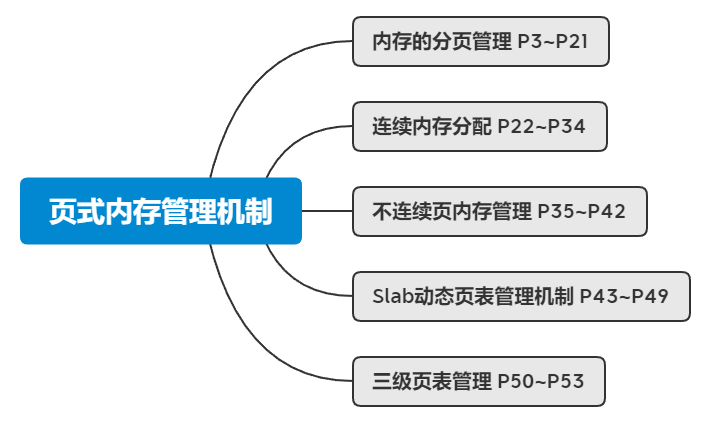
**课程时间线：**



**课外参考读物：**

《操作系统导论》

**知识框图：**

****

PPT讲稿：

3. 上一讲我们说了进程的分段，一般分为三段，代码段、堆段、栈段。通过这种方式创建的进程有一个问题，就是容易产生内存碎片。举一个例子，一个没有划线的停车场，大家去停车，什么车型都有，小轿车、suv，大卡车、公交车，挖掘机。大家都停的很规矩，挨着停，非常紧凑，但是不好管理。停车场管理员想去找一辆车是很费劲的。另一个停车场划了线，每一个车位都是一样大的，要求每一个车都停车到位。如果车特别大，也可以占两个车位。然后每一个车位都有一个唯一的编号，虽然空间上有一点浪费。但是管理起来非常方便，管理员只要知道车位编号就可以了。所以这个就是页表最根本的思想。对内存进行固定大小区域的划分。然后往里面去塞数据和代码。
4. 看图。左边是一个虚拟的地址空间，一共64个字节，我规定，一个页的长度是16字节，所以这个地址空间可以平分为4个页。对应的，在物理内存里也有同样大小的4个页。要注意，不光是虚拟内存按每16字节划分一个页，物理地址也是这份划分的。左边的叫做虚拟页，右边的叫做页帧。在我们真实的操作系统上，一般页的大小都是4k。也就是在内存的世界里，每一个车位的宽度都是4k。你的车再小，一辆自行车，也得占4k空间。你的车很长，一辆火车，可以占很多很多车位，但最终所占用的空间也还是4k的整数倍。其实按照固定大小的空间去管理一个更大的空间是一种很常见的策略。比如说我们国家的铁路系统，他们把铁轨分成很多份，一份可能是5公里，后台调度系统就要求，任何时刻，任意的一段铁路，最多只能有一趟列车。基于这样一个基本原理，我再去设计后面更复杂的需求。比如前面大雪封路了，前面列车停下。后面的列车通过数据中心，发现前面一段的铁路被占用了，然后自己就在当前段停了待命。后面的列车都在自己的车辆段排队。这样就很高效。如果我设计一个系统，特别细化，要求每辆列车离前方列车最近不能超过3km，那就意味这总是要进行一些额外计算，然后管理起来也非常混乱。  
   说回内存。我们从虚拟空间到真实空间每一个页都是有一一对应关系的，虚拟页可能存在于物理内存中的任何一帧，取决于操作系统的分配。从虚拟页到页帧的映射，每个进程都有一个专门的数据结构来管理，我们称之为页表。页表是一个数组，每一个元素称之为页表项。每一个页表项记录着该虚拟地址到物理地址的转换，以及一些其他的相关信息。
5. 具体看一下虚拟地址是怎么转换成物理地址的。例子中是64个字节，一共4个虚拟页。所以需要2的6次方才能完整表示每一个地址。一共就是6位。前两位我们用来作为页的序号，刚好4页。后面四位是对应页的偏移量。这个和上一节讲的分段是一个意思。举一个例子，看这条指令 movl <virtual address>, %eax 。将一个虚拟地址加载到寄存器。我们假设这个虚拟地址是21。按位表示就是010101。前两位我们叫虚拟页号，VPN。后面四位是偏移量。01转换为十进制是1。于是就去页表里查找第二项，拿出来我们叫页表项，页表项的前几位表示的就是页帧编号。这样那PFN和偏移量放在一起就组成了真实的物理地址。地址转换就算完成了。
6. 刚才说到了页表项，那么页表项里究竟都是写什么东西呢？我们以x86的页表项为例，一共32位。第12到31位是用来表示PFN的。0到8位是一些标志位。P存在位，表示当前的地址是存在于内存还是磁盘上。有的同学可能会奇怪，内存地址怎么会在磁盘上呢？这个涉及到内存交换的问题，以后我们会讲。R/W表示表示当前地址是否可写。比如代码段就肯定是不可写，堆、栈根据具体情况决定能不能写。U/S表示的是当前用户是否是超级用户，属于是权限的控制，权限低就没办法访问当前内存。A表示访问位，用来表示当前页是否被访问过，是比较关键和重要的位，之后我们也会详细展开。D是dirty的意思，叫脏位。表示当前页是否被修改过。这四位G\PAT\PCD\PWT都是和缓存相关的位。
7. 有关这个分页的具体操作，我贴上代码来给大家过一遍，这样能有一个直观的理解。一个最简单的指令 movl 21,%eax 将虚拟地址21的数据放入寄存器。 那这里系统需要获得真实地址117。那么为了获得物理地址，必须要查询相应的页表，那为何查询页表，我还必须知道页表的地址。反过来按正序来走一遍流程。假设我们有一个页表基址寄存器。第一步查到页表基址。然后检索页表，提取页表项，完成物理地址转换，从物理地址获取数据并存入寄存器。
   * 1. 1 // Extract the VPN from the virtual address
     2. 2 VPN = (VirtualAddress & VPN\_MASK) >> SHIFT
     3. 3
     4. 4 // Form the address of the page-table entry (PTE)
     5. 5 PTEAddr = PTBR + (VPN \* sizeof(PTE))
     6. 6
     7. 7 // Fetch the PTE
     8. 8 PTE = AccessMemory(PTEAddr)
     9. 9
     10. 10 // Check if process can access the page
     11. 11 if (PTE.Valid == False)
     12. 12 RaiseException(SEGMENTATION\_FAULT)
     13. 13 else if (CanAccess(PTE.ProtectBits) == False)
     14. 14 RaiseException(PROTECTION\_FAULT)
     15. 15 else
     16. 16 // Access is OK: form physical address and fetch it
     17. 17 offset = VirtualAddress & OFFSET\_MASK
     18. 18 PhysAddr = (PTE.PFN << PFN\_SHIFT) | offset
     19. 19 Register = AccessMemory(PhysAddr)
   1. 我们直接看代码：第二行获取虚拟地址的VPN。
   2. 第五行据页表寄存器的值组合得到页表项的地址。
   3. 第八行提取出当前的页表项
   4. 最后一段先做一些安全性的判别。
   5. 成功后获取偏移值，通过和物理基址（通过页表项就可以提取出PFN的值）的组合形成物理地址。
   6. 最后读取物理地址的数据写入寄存器。。
   7. 所以大家可以看到，一个最简单的从内存加载数据写入寄存器才操作就需要这么复杂的一套流程，所以分页带来高效管理的同时也会导致内存读取的速度比较慢。
8. 内存分页是非常重要的一个知识点，我们再从内存的角度来追踪一下。我们看这样一段代码：
   * 1. int array[1000];
     2. ...
     3. for (i = 0; i < 1000; i++)
     4. array[i] = 0;
   1. 一个简单地循环，把一个1000长度的数组全部赋值为0。
      1. 1024 movl $0x0,(%edi,%eax,4)
      2. 1028 incl %eax
      3. 1032 cmpl $0x03e8,%eax
      4. 1036 jne 0x1024
   2. 翻译成汇编就是这四行
   3. 第一行是将0值移动到虚拟内存地址。
   4. 第二行是增加数组索引。
   5. 第三行是将数组索引和1000对比。一样的话就结束循环
   6. 第四行是跳转到第一行，循环执行。
9. 看下内存角度的具体操作。首先，我们拿到虚拟地址1024，也就是1k。将1024进行拆解，得到VPN和偏移量。拿VPN去页表里面找到对应的页表项，取出PFN，把PFN和偏移量组合，得到当前指令的真实物理地址，取出mov指令。这里对应的是左下角第一个小黑块。mov是移动指令，我们看一下这个指令 movl $0x0, (%edi,%eax,4)。意思是将0移动到数组。edi是数组的基址，eax是数组索引，4是单个数组元素的大小，也就是int，4个字节。所以这个指令接下来要去页表里面找到数组的物理地址，然后将数值写入数组索引所在的内存地址。然后继续通过虚拟地址转换物理地址下一条指令incl %eax 将数组的索引值加1.那么下一次往数组写数据的时候就是1位置了。在图中也可以看到inc的地址比前一条增加了一点。第三条是一个判别循环是否完成，第四年是跳转到开始，进行下一轮循环。我们看下面这一行，一共4条指令，实在反复的循环执行的。中间的是数组的地址，随着数据的存入，逐渐的增加。最上面是页表的访问情况，反复的在代码和数据段进行跳转，这里数据少，只显示了5次的循环。到后面上面的这个页表会变成40、41、42。一直到循环结束为止。
10. 通过这一个例子，我们可以对内存的分页有一个了解。整体来说，和上一节讲的内容相比，分页按照固定大小来整理内存，显得干净利落。页表项也提供了更加丰富的判别项。
11. 我们看到通过这样频繁的在页表里查找虚拟页号到物理帧号的映射是非常繁琐的过程，需要消耗大量的资源。但其实反复都在访问那么两个页表，造成了很大的浪费。为了解决这个问题，科学家们提出了一种快速地址转换方法。想法非常简单，就是把刚用过的地址转换存起来，放在一个列表里面。然后我访问下一条指令的时候，先在这个列表里查看有没有，有的话直接拿来用，没有的话我再去查页表，查完了放到这个列表里。这个列表就是一个缓存器。全称叫做地址转换旁路缓冲存储器（translation-lookaside buffer)简称TLB。
12. 贴一段代码直观的感受一下。

1 VPN = (VirtualAddress & VPN\_MASK) >> SHIFT

2 (Success, TlbEntry) = TLB\_Lookup(VPN)

3 if (Success == True) // TLB Hit

4 if (CanAccess(TlbEntry.ProtectBits) == True)

5 Offset = VirtualAddress & OFFSET\_MASK

6 PhysAddr = (TlbEntry.PFN << SHIFT) | Offset

7 Register = AccessMemory(PhysAddr)

8 else

9 RaiseException(PROTECTION\_FAULT)

10 else // TLB Miss

11 PTEAddr = PTBR + (VPN \* sizeof(PTE))

12 PTE = AccessMemory(PTEAddr)

13 if (PTE.Valid == False)

14 RaiseException(SEGMENTATION\_FAULT)

15 else if (CanAccess(PTE.ProtectBits) == False)

16 RaiseException(PROTECTION\_FAULT)

17 else

18 TLB\_Insert(VPN, PTE.PFN, PTE.ProtectBits) //插入TLB

19 RetryInstruction()

上来先获取虚拟页号VPN，然后去TLB看一眼，里面有没有缓存项。假设第一次访问，那肯定没有，于是跳到第10行。老老实实的从页表进行地址转换。然后没啥问题了，跳转到第18行，插入TLB中。然后下一次再取地址的时候，TLB命中，跳到第3行。直接取PFN，加上偏移获得物理地址，完成。这里有一个很关键的问题，就是TLB的命中率有多高。我们肯定希望越高越好，这样就不用频繁的去浪费时间去页表里查询。

1. 来具体跟踪一小段TLB的操作。假设有一段8位的地址空间，平均分成16份，每份是一个虚拟页，每个页的大小是16字节。每一行都有一个虚拟页号，从0到15。横坐标就是字节偏移量。假设我们有一个整型数组a，里面一共有十个元素。第一个元素存储在了虚拟地址100的地方，每一个元素是4字节，如图就是这样存储的。然后我们写一个循环，遍历这个数组。看下TLB是怎么工作的。在循环开始前，TLB是空的，所以a[0]的物理地址是从页表获取的，然后放入了TLB中。TLB列表项里面存放了VPN和FPN。然后继续循环，读取a[1]，这时候发现vpn是06，先查找一下TLB，发现，有这一项，然后就可以直接把这个FPN值拿过来用，转换为物理地址。这样就是TLB命中，减少了地址转换的流程和时间，提高了效率。所以叫快速地址转换。那么TLB 命中率越高，我对内存地址转换的效率就越高。我们计算一下命中率，10次访问，7次命中，就是70%。实际操作中，我们希望这个值越大越好。我们的例子中每页的大小是16字节，如果扩大一倍为32字节，命中率就能达到80%，再扩大一倍就能达到90%。当然也不能为了追求命中率去一味的追求大的页，还需要考虑其他因素。一般来说我们操作系统用的都是4k大小的页。
2. 【TLB内容】和前面说的页表项一样，TLB的每一项也存在很多的标志位。比如有效位、保护位、脏位等等。为什么需要这些其他位，因为实际运行中可能出现的状况有很多，每个TLB项的状态也都不同，必须要加以区分，来保证我拿到的数据是没问题的。以上下文切换为例。假设系统中运行有两个进程，P1和P2，都分别在TLB中有虚拟地址映射。P1的VPN是10，映射到PFN是100，P2的VPN也是10，映射到的地址是170。那么问题来了，我的操作系统怎么转换呢？最简单的方法就是每次切换进程的时候清空TLB，然后再进程运行的时候把我刚才讲的那一套来一遍，换个进程再来一遍，哪怕说我从P1换到P2，再从P2换回P1，我也要当P1是一个新进程来填充TLB。这样的话效率明显是很低。为了解决这个问题，TLB有一项叫做ASID，这个类似于PID，也是每个进程的唯一标识符。在通过VPN查找FPN的时候呢，也检查一下ASID。如此就做到了一个TLB列表可以共享给多个进程使用。
3. 这个列表呢，是一个真实的操作系统TLB项。VPN,PFN没什么好说的。这里的ASID我们看到是8位的，其实这个用PID也没有问题，但是PID太长了，32位，像这样一个64位长度的TLB项都装不下。G位，意思是global，全局位，如果是1，就表示我可以忽略ASID，别的进程也能访问。D代表脏位，表示是否被修改过。C表示一致性位，V是有效位。不同的架构下页表项的定义也不同。TLB和任何其他缓存一样，都是有固定大小的，当你操作系统运行的进程多了，TLB总会饱和，这个时候再出现新的映射的时候，我们就要对一些缓存进行替换。那么替换谁，不替换谁就很有讲究，产生了很多方法。比方说我去替换最近用的比较少的缓存，或者去随机的替换一个缓存。这里的算法也很多，就不展开讲了。
4. 前面讲到分页有两大问题，一个是太慢，一个是太大。太慢的问题我们通过TLB解决了，现在就来说说太大的问题。我们具体看一下页表为什么大。一个例子，假设我们有一个14位的虚拟地址空间，也就是16k的大小。一个页的大小是1k，那么就是一共16个页。但其实我只有四个页是有数据的，分别是代码段1页，堆段1页，栈段2页。然后通过页表映射到物理内存地址。我们把页表展开看一下。这个页表必须将VPN和PFD一一对应，所以页表的大小就是16。但这一个数组我们能看到，就4项是有效的剩下的全是无效的，非常的浪费空间。要是32位的地址空间，甚至64位的地址空间按到这样会产生多大的空间浪费呢？要知道页表也是要存到内存里的。内存里面寸土寸金，还是要存一些有用的东西。
5. 【混合方法：分页+分段】那怎么办呢？前面我们讲过分段，还记得吧，代码段、堆段、栈段。之前说分段可以有效的解决物理内存连续占用的问题。那这个思路能不能拿来解决页表占用内存过大的问题呢？答案是可以的。其实非常简单，把上面的页表拆成三个页表就可以了。然后每一个页表配一组基址加界限，就能保证没用的地址不占用页表空间，大幅减少了页表的空间占用。
6. 另一种解决方案叫做多级页表。多级页表不分段，但思路和分段的思路很像。多级页表首先也将我们的普通页表分割成很多份。比如说啊，一个原始的线性页表大小是1000。然后在这个系统里面规定，单页的大小是100。所以呢我就把我的线性页表分成了10份。等于是10个子页表。这10个子页表是可以零散的存在内存中的，这样就解决了一个问题，不需要大段的连续内存，也可以存储大页表。那么为了记录着10个子页表在内存中的位置。我又额外有了一个页目录。这样我去通过虚拟地址查找真实地址的时候，首先查找页目录，看下我的虚拟地址在10个子页表中的哪一个，然后再到子页表中具体查找对应的物理地址。这就是一个多级的概念。这里的例子是一个最简单的二级页表，实际操作中，可以有三级、四级等等。级数越多就越复杂，但页表也可以拆分的更散。
7. 有了页目录以后，我们页表的可玩性就很高了。我们这一张图可以看到，其实10个子页表中，只有两个子页表是有有效数据的，其余的8个子页表，都是废的，没有作用，但还占着宝贵的内存空间。所以我们只要在页目录里面加一个有效项，标记一下，指向的子页表是不是空，如果不是空，就正常存放在内存中，如果是空，我就不分配内存。等到不是空的时候再分配也来得及。这样一操作，就让页表的占用空间大幅的减少。解决了前面说的页表占用空间大的问题。事实上很多现代的操作系统，都用的是多级页表来管理内存。原来我分配一个这样大的页表，需要10页的地址空间。现在呢需要三页。一页用来存放页目录，两页用来存放子页表，大大减少了空间占用。
8. 有的时候两级页表可能不太够用。为什么呢？我们的目标是把页表的每一个部分都能放入一个页内。因为实际操作系统中，都是32位或者64位的地址。原始的线性页表大小可能非常大。还是按前面的例子，如果一个原始线性页表大小是20000，单页大小还是100。那么页目录的大小就是20000除以100，等于200，就会导致一页存不下我们的页目录。所以我们可以页目录拆成两份，也就是一级页目录、二级页目录的结构。所以实际的系统中都是很多级的页表。
10. 连续分配方式，是指为一个用户程序分配一个连续的内存空间，该连续内存空间指的的是物理内存，主要有以下四种分配方式:单一连续分配、固定分区分配、动态分区分配和动态重定位分区分配
11. 单一连续分配是指内存可以分为系统内存区和用户区两部分，系统区供OS使用，用户区供用户使用；单一连续分配，就是把用户区当成了一个整体用；所以，该内存管理方式只适用于单用户单任务的操作系统
12. 固定分区分配又称定长分区或静态分区模式，是满足多道程序设计需要的最简单的存储管理技术。基本思想是给进入主存的用户作业划分一块连续存储区域，把作业装入该连续存储区域，若有多个作业装入主存，则它们可并发执行；难点是程序可能太大而不能放到一个分区中，内存的利用率很低；由于被装入的数据块小于分区大小，从而导致分区内部有浪费现象，成为“内部碎片”；优点是实现简单，只需要极少的操作系统开销；缺点是有内部碎片，对内存的使用不充分，活动进程的最大数目是固定的
13. 动态分区存储管理不是预先把内存中的用户区域划分成若干固定分区，而是在作业要求装入内存时，根据用户作业的大小和当时内存空间使用情况决定是否为该作业分配一个分区。因此分区大小不是预先固定的，而是按作业需求量来划分的；分区的个数和位置也不是预先确定的。它有效地克服了固定分区方式中，由于分区内部剩余内存空置造成浪费的问题
14. 系统初启时，整个用户区可看作一个大的空闲区。当作业要求装入时，根据作业对内存需求量，从空闲区中划出一个与作业大小一致的分区来装入该作业，剩余部分仍为空闲区。当空闲区能满足需求时（即空闲区长度>=作业长度），作业可装入；否则，作业暂时不能装入
15. 装入内存的作业执行结束后，所占分区被收回成为一个空闲区，这个空闲区又可用于装入其他作业。随着作业不断装入和撤离，内存空间被分成许多分区，有的被作业占用，有的空闲。可见内存中空闲区数目和大小是在不断变化
16. 系统设置一张“ 空闲区表 ”，用来记录空闲区的起始地址和长度。当有作业要装入内存时，在空闲区表中找一找“ 未分配 ”的栏目，从中找出一个能容纳作业的空闲区。若空闲区大于作业的长度时则被分成两部分，一部分分配给作业；另一部分仍作为空闲区登记在表中。若找到的空闲区等于作业长度时,分配后该栏目状态改为“空”状。当有作业撤离收回所占分区后，应把收回区域的起始地址和长度登记在状态为“空”的栏目中，且将状态改为“未分配”。如果收回的区域正好和某一空闲区相邻，则应将其合并成一个分区后登记
17. 上图给出了存储空间的分配方式，我们可以看到所有空闲去都有在下面的空闲区表中进行了记录
18. 对于动态分区分配的内存分配算法，最常用到的是最先适应分配算法。每次分配总是顺序查找空闲区表，找到能满足长度要求的空闲区就分配。优点是实现简单，缺点是可能将大的空闲区分割成许多小的空闲区，形成许多不连续的“碎片”。碎片长度可能不能满足作业要求，降低了内存利用率。改进方法是把空闲区按地址顺序从小到大登记在空闲区表中，有利于大作业。问题是归还空区时须按地址插入表中适当位置。
19. 而最优适应分配算法按作业要求从所有空闲区中挑选一个能满足要求的最小空闲区，这样保证不去分割一个更大的区域，使装入大作业时比较容易得到满足。实现办法是将空闲区按长度以递增次序登记在表中，分配时按空闲区表顺序查找即可。缺点是可能碎片更小而无法使用。回收时也要按长度扦入
20. 最坏适应分配算法按作业要求从所有空闲区中挑选一个能满足要求的最小空闲区，这样保证不去分割一个更大的区域，使装入大作业时比较容易得到满足。实现办法是将空闲区按长度以递增次序登记在表中，分配时按空闲区表顺序查找即可。缺点是可能碎片更小而无法使用。回收时也要按长度扦入
21. 而动态重定位分区分配是把不同程序，且在内存中地址不连续的想办法让他们连续。例如，两个程序A、B，在内存中的位置不连续，即：A和B中间有空闲内存，我就把B和那部分的空闲区对换，为了实现改过程序我们增加了一个硬件——重定位寄存器。虽然这种分区非常的好，但是，要真正的实现起来需要为此付出很大的开销，为了解决这个问题，人们又发明了一种分配方式——分页
22. 接下来我们介绍不连续页内存管理
23. 前面的几种存储管理方法，为进程分配的空间是连续的，使用的都是物理地址。如果允许将进程分散到许多不连续的空间，就可以避免内存紧缩，减少碎片。通过引入进程的逻辑地址，把进程地址空间与实际存储空间分离，增加存储管理的灵活性。根据分配时所采用的基本单位不同，可将离散分配的管理方式分为三种：页式存储管理；段式存储管理和段页式存储管理；
24. 将逻辑地址空间划分为固定大小的页(page)，而物理内存划分为同样大小的页框(page frame)。程序加载时，可将任意一页放人内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要CPU的硬件支持，来实现逻辑地址和物理地址之间的映射。在页式存储管理方式中地址结构由两部构成，前一部分是页号，后一部分为页内地址w（位移量）
25. 他的优缺点同样明显，优点是没有外碎片，每个内碎片不超过页大小；一个程序不必连续存放；便于改变程序占用空间的大小(主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长)。缺点是要求程序全部装入内存，没有足够的内存，程序就不能执行
26. 每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序。进程页表完成逻辑页号(本进程的地址空间)到物理页面号(实际内存空间，也叫块号)的映射
27. 整个系统有一个物理页面表，描述物理内存空间的分配使用状况，其数据结构可采用位示图和空闲页链表。对于位示图法，即如果该页面已被分配，则对应比特位置1，否置0。
28. 整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换也可以结合到各进程的PCB(进程控制块)里。我们可以看图中，就是一个典型的请求表，记录了进程信息和请求页面信息。
29. 在页式系统中，指令所给出的地址分为两部分：逻辑页号和页内地址。CPU中的内存管理单元(MMU)按逻辑页号通过查进程页表得到物理页框号，将物理页框号与页内地址相加形成物理地址
31. 以页为最小单位分配内存对于内核管理系统物理内存来说的确比较方便，但内核自身最常使用的内存却往往是很小（远远小于一页）的内存块。存放文件描述符、进程描述符、虚拟内存区域描述符等行为所需的内存都不足一页。为了满足内核对这种小内存块的需要，Linux系统采用了slab分配器的技术。Slab的实现相当复杂，但原理不难，其核心思想就是“存储池”的运用。内存片段被看作对象，当被使用完后，并不直接释放而是被缓存到“存储池”里，留做下次使用。这无疑避免了频繁创建与销毁对象所带来的额外负载
32. Slab技术不但避免了内部碎片带来的不便，引入Slab分配器的主要目的是为了减少对伙伴系统分配算法的调用次数，频繁分配和回收必然会导致内存碎片，难以找到大块连续的可用内存；而且可以很好利用硬件缓存提高访问速度
33. 在最高层是 cache\_chain，这是一个 slab 缓存的链接列表。这对于 best-fit 算法非常有用，可以用来查找最适合所需要的分配大小的缓存（遍历列表）。cache\_chain 的每个元素都是一个 kmem\_cache 结构的引用（称为一个 cache）。它定义了一个要管理的给定大小的对象池。
34. 每个缓存都包含了一个 slabs 列表，这是一段连续的内存块（通常都是页面）。存在 3 种 slab：完全分配的、部分分配的和空 slab。注意 slabs\_empty 列表中的 slab 是进行回收（reaping）的主要备选对象。正是通过此过程，slab 所使用的内存被返回给操作系统供其他用户使用。slab 列表中的每个 slab 都是一个连续的内存块（一个或多个连续页），它们被划分成一个个对象。这些对象是从特定缓存中进行分配和释放的基本元素。注意 slab 是 slab 分配器进行操作的最小分配单位，因此如果需要对 slab 进行扩展，这也就是所扩展的最小值。通常来说，每个 slab 被分配为多个对象
35. 这里列出了典型的SLAB的API函数，包括缓存的创建、销毁、分配和释放等操作。
36. Slab分配模式把对象分组放进缓冲区，因为缓冲区的组织和管理与硬件高速缓存的命中率密切相关，因此，Slab缓冲区并非由各个对象直接构成，而是由一连串的“大块（Slab）”构成，而每个大块中则包含了若干个同种类型的对象。一般而言，对象分两种，一种是大对象，一种是小对象。所谓小对象，是指在一个页面中可以容纳下好几个对象的那种。Linux内核中把小于512字节的对象叫做小对象
37. 接下来我们介绍三级页表管理
38. 这是Linux典型的三级分页方式，我们上一章页有介绍。主要包括了全局页目录、中间页目录表和页表，通过三级分页寻址和页内偏移，将线性地址转换为物理地址
39. 页全局目录(Page Global Directory)：即 pgd，是多级页表的抽象最高层。每一级的页表都处理不同大小的内存。每项都指向一个更小目录的低级表，因此pgd就是一个页表目录。当代码遍历这个结构时（有些驱动程序就要这样做），就称为是在遍历页表；页中间目录 (Page Middle Directory)：即pmd，是页表的中间层。在 x86 架构上，pmd 在硬件中并不存在，但是在内核代码中它是与pgd合并在一起的；页表条目 (Page Table Entry)：即pte，是页表的最低层，它直接处理页，该值包含某页的物理地址，还包含了说明该条目是否有效及相关页是否在物理内存中的位
40. 分段可以给每个进程分配不同的线性地址空间，分页可以把同一线性地址空间映射到不同的物理空间。与分段相比，Linux更喜欢分页方式，因为当所有进程使用相同的段寄存器值时，内存管理变得更简单，也就是说它们能共享同样的一簇线性地址；为了兼容绝大多数的CPU，RISC体系架构对分段的支持很有限，比如ARM架构的CPU中的MMU单元通常只支持分页，而不支持分段